内核通知链

linux

Linux内核中各个子系统相互依赖，当其中某个子系统状态发生改变时，就必须使用一定的机制告知使用其服务的其他子系统，以便其他子系统采取相应的措施。为满足这样的需求，内核实现了事件通知链机制（notificationchain）

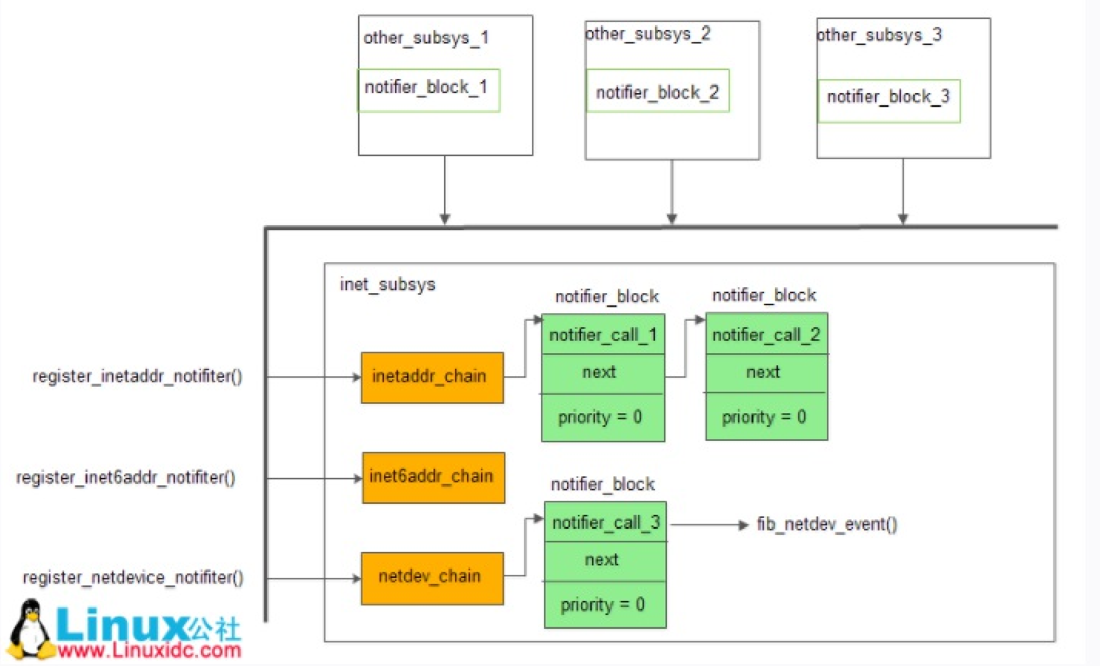

图 1 内核通知链

Linux的网络子系统一共有3个通知链：表示ipv4地址发生变化时的inetaddr_chain；表示ipv6地址发生变化的inet6addr_chain；还有表示设备注册、状态变化的netdev_chain。

在这些链中都是一个个notifier_block结构：

```
struct notifier_block {
      int (*notifier_call)(struct notifier_block *, unsigned long, void *);
      struct notifier_block *next;
      int priority;
};
```

其中，

1. notifier_call：当相应事件发生时应该调用的函数，由被通知方提供，如other_subsys_1；
2. notifier_block *next：用于链接成链表的指针；
3. priority：回调函数的优先级，一般默认为0。

内核代码中一般把通知链命名为xxx_chain，xxx_nofitier_chain这种形式的变量名。围绕核心数据结构notifier_block，内核定义了四种通知链类型：

1. 原子通知链（ Atomic notifier chains ）：通知链元素的回调函数（当事件发生时要执行的函数）在中断或原子操作上下文中运行，不允许阻塞。对应的链表头结构：

```
struct atomic_notifier_head {
        spinlock_t  lock;
        struct  notifier_block *head;
};
```

2. 可阻塞通知链（ Blocking notifier chains ）：通知链元素的回调函数在进程上下文中运行，允许阻塞。对应的链表头：

```
struct  blocking_notifier_head {
        struct  rw_semaphore  rwsem;
        struct  notifier_block  *head;
```

};

3. 原始通知链（ Raw notifierchains ）：对通知链元素的回调函数没有任何限制，所有锁和保护机制都由调用者维护。对应的链表头：

网络子系统就是该类型，通过以下宏实现head的初始化

```
static RAW_NOTIFIER_HEAD(netdev_chain);
#define RAW_NOTIFIER_INIT(name)     {       \
              .head= NULL }
#define RAW_NOTIFIER_HEAD(name)         \       //调用他就好了
struct raw_notifier_head name =         \
                    RAW_NOTIFIER_INIT(name)
```

即：

```
struct raw_notifier_head netdev_chain = {
          .head = NULL;
}
```

而其回调函数的注册，比如向netdev_chain的注册函数：register_netdevice_notifier。

```
struct  raw_notifier_head {
        struct  notifier_block  *head;
};
```

4. SRCU 通知链（ SRCU notifier chains ）：可阻塞通知链的一种变体。对应的链表头：

```
struct  srcu_notifier_head {
        struct  mutex mutex;
        struct  srcu_struct  srcu;
        struct  notifier_block  *head;
};
```

1.3. 运行机理

被通知一方（other_subsys_x）通过notifier_chain_register向特定的chain注册回调函数，并且一般而言特定的子系统会用特定的notifier_chain_register包装函数来注册，比如路由子系统使用的是网络子系统的：register_netdevice_notifier来注册他的notifier_block。

1.3.1. 向事件通知链注册的步骤

1. 申明struct notifier_block结构
2. 编写notifier_call函数
3. 调用特定的事件通知链的注册函数，将notifier_block注册到通知链中

如果内核组件需要处理够某个事件通知链上发出的事件通知，其就该在初始化时在该通知链上注册回调函数。

1.3.2. 通知子系统有事件发生

inet_subsys是通过notifier_call_chain来通知其他的子系统（other_subsys_x）的。

notifier_call_chain会按照通知链上各成员的优先级顺序执行回调函数（notifier_call_x）；回调函数的执行现场在notifier_call_chain进程地址空间；其返回值是NOTIFY_XXX的形式，在include/linux/notifier.h中：

```
#define NOTIFY_DONE             0x0000        /* 对事件视而不见 */
#define NOTIFY_OK           0x0001        /* 事件正确处理 */
#define NOTIFY_STOP_MASK  0x8000        /*由notifier_call_chain检查，看继续调用回调函数，还是停止，_BAD和_STOP中包含该标志 */
#define NOTIFY_BAD        (NOTIFY_STOP_MASK|0x0002)      /*事件处理出错，不再继续调用回调函数 */
/*
 *Clean way to return from the notifier and stop further calls.
 */
#define NOTIFY_STOP             (NOTIFY_OK|NOTIFY_STOP_MASK)    /*  回调出错，不再继续调用该事件回调函数
```

*/

notifier_call_chain捕获并返回最后一个事件处理函数的返回值；注意：notifier_call_chain可能同时被不同的cpu调用，故而调用者必须保证互斥。

1.3.3. 事件列表

对于网络子系统而言，其事件常以NETDEV_XXX命名；描述了网络设备状态（dev->flags）、传送队列状态（dev->state）、设备注册状态（dev->reg_state），以及设备的硬件功能特性（dev->features）：

include/linux/notifier.h中

```
/* netdevice notifier chain */
#define NETDEV_UP  0x0001  /* 激活一个网络设备 */
#define NETDEV_DOWN  0x0002f /* 停止一个网络设备，所有对该设备的引用都应释放 */
#define NETDEV_REBOOT       0x0003       /* 检查到网络设备接口硬件崩溃，硬件重启 */
#define NETDEV_CHANGE       0x0004  /* 网络设备的数据包队列状态发生改变 */
#define NETDEV_REGISTER  0x0005  /*一个网络设备事例注册到系统中，但尚未激活 */
#define NETDEV_UNREGISTER       0x0006       /*网络设备驱动已卸载 */
#define NETDEV_CHANGEMTU       0x0007  /*MTU发生了改变 */
#define NETDEV_CHANGEADDR     0x0008  /*硬件地址发生了改变 */
#define NETDEV_GOING_DOWN  0x0009  /*网络设备即将注销，有dev->close报告，通知相关子系统处理 */
#define NETDEV_CHANGENAME  0x000A  /*网络设备名改变 */
#define NETDEV_FEAT_CHANGE     0x000B  /*feature网络硬件功能改变 */
#define NETDEV_BONDING_FAILOVER 0x000C  /*    */
#define NETDEV_PRE_UP         0x000D  /*    */
#define NETDEV_BONDING_OLDTYPE  0x000E                /*    */
#define NETDEV_BONDING_NEWTYPE  0x000F       /*    */
```

linux

1.4. 简单一例：

通过上面所述，notifier_chain机制只能在内核个子系统间使用，因此，这里使用3个模块：test_notifier_chain_0、test_notifier_chain_1、test_notifier_chain_2；当 test_notifier_chain_2通过module_init初始化模块时发出事件TESTCHAIN_2_INIT；然后 test_notifier_chain_1作出相应的处理：打印 test_notifier_chain_2正在初始化。

```
/* test_chain_0.c ：0. 申明一个通知链；1. 向内核注册通知链；2. 定义事件； 3. 导出符号，因而必需最后退出*/
#include <linux/notifier.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h> /* printk() */
#include <linux/fs.h> /* everything() */
#define TESTCHAIN_INIT 0x52U
static RAW_NOTIFIER_HEAD(test_chain); 定义通知链链表头

/* define our own notifier_call_chain */
static int call_test_notifiers(unsigned long val, void *v) 只是提供了函数接口，在这个内核模块中并没有调用
{
    return raw_notifier_call_chain(&test_chain, val, v);  定义自己的通知链
}
EXPORT_SYMBOL(call_test_notifiers);

/* define our own notifier_chain_register func */
static int register_test_notifier(struct notifier_block *nb) 只是提供了函数接口，在这个内核模块中并没有调用
{
```

```
    int err;
    err = raw_notifier_chain_register(&test_chain, nb);  注册通知链
    if(err)
        goto out;
    out:
        return err;
}

EXPORT_SYMBOL(register_test_notifier);
static int __init test_chain_0_init(void)
{
    printk(KERN_DEBUG "I'm in test_chain_0\n");
    return 0;
}

static void __exit test_chain_0_exit(void)
{
    printk(KERN_DEBUG "Goodbye to test_chain_0\n");
    // call_test_notifiers(TESTCHAIN_EXIT, (int *)NULL);
}
MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("fishOnFly");
module_init(test_chain_0_init);
module_exit(test_chain_0_exit);

/* test_chain_1.c ：1. 定义回调函数；2. 定义notifier_block；3. 向chain_0注册notifier_block；*/
#include <linux/notifier.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h> /* printk() */
#include <linux/fs.h> /* everything() */
extern int register_test_notifier(struct notifier_block *nb);
#define TESTCHAIN_INIT 0x52U
/* realize the notifier_call func */
int test_init_event(struct notifier_block *nb, unsigned long event,void *v)
{
    switch(event){
        case TESTCHAIN_INIT: 如果收到对应的事件
            printk(KERN_DEBUG "I got the chain event: test_chain_2 is on the way of init\n");
            break;
        default:
            break;
    }
    return NOTIFY_DONE;
}

/* define a notifier_block */
static struct notifier_block test_init_notifier = {
```

```
    .notifier_call = test_init_event,  定义回调函数，主要就是定义notifiler_block结构体
};

static int __init test_chain_1_init(void)
{
    printk(KERN_DEBUG "I'm in test_chain_1\n");
    register_test_notifier(&test_init_notifier);  注册notifier_block 到通知链中
    return 0;
}

static void __exit test_chain_1_exit(void)
{
    printk(KERN_DEBUG "Goodbye to test_clain_l\n");
}
MODULE_LICENSE("GPL");
MODULE_AUTHOR("fishOnFly");
module_init(test_chain_1_init);
module_exit(test_chain_1_exit);



/* test_chain_2.c：发出通知链事件*/
#include <linux/notifier.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h> /* printk() */
#include <linux/fs.h> /* everything() */
extern int call_test_notifiers(unsigned long val, void *v);
#define TESTCHAIN_INIT 0x52U
static int __init test_chain_2_init(void)
{
    printk(KERN_DEBUG "I'm in test_chain_2\n");
    call_test_notifiers(TESTCHAIN_INIT, "no_use"); 定义通知链
    return 0;
}
static void __exit test_chain_2_exit(void)
{
    printk(KERN_DEBUG "Goodbye to test_chain_2\n");
}
MODULE_LICENSE("GPL v2");
MODULE_AUTHOR("fishOnFly");
module_init(test_chain_2_init);
module_exit(test_chain_2_exit);

# Makefile
# Comment/uncomment the following line to disable/enable debugging
# DEBUG = y

# Add your debugging flag (or not) to CFLAGS
```

```
ifeq ($(DEBUG),y)
DEBFLAGS = -O -g -DSCULL_DEBUG # "-O" is needed to expand inlines
else
DEBFLAGS = -O2
endif

ifneq ($(KERNELRELEASE),)
# call from kernel build system
obj-m := test_chain_0.o test_chain_1.o test_chain_2.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
modules:
$(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif

clean:
rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
depend .depend dep:
$(CC) $(CFLAGS) -M *.c > .depend

ifeq (.depend,$(wildcard .depend))
include .depend
endif
```

```
[wang2@iwooing: notifier_chian]$ sudo insmod./test_chain_0.ko
[wang2@iwooing: notifier_chian]$ sudo insmod./test_chain_1.ko
[wang2@iwooing: notifier_chian]$ sudo insmod./test_chain_2.ko
[wang2@iwooing: notifier_chian]$ dmesg
[ 5950.112649] I'm in test_chain_0
[ 5956.766610] I'm in test_chain_1
[ 5962.570003] I'm in test_chain_2
[ 5962.570008] I got the chain event: test_chain_2 is on the way of init
[ 6464.042975] Goodbye to test_chain_2
[ 6466.368030] Goodbye to test_clain_1
[ 6468.371479] Goodbye to test_chain_0
```

主要编写流程：首先用RAW_NOTIFIER_HEAD(一共四种方式创建)创建一个通知链的头部，然后一个模块定义struct notifier_block结构体，在该结构体中注册回调函数。调raw_notifier_chain_register函数将notifilter_block注册到通知链中。同时定义一个操作码，在回调函数中判断传入的操作码。在另一个模块中直接调用raw_notifier_call_chain(其实最终调的是notifier_call_chain，只是raw_notifier_call_chain对notifier_call_chain进行了封装)函数传入操作码即可。

最近在看《深入理解Linux网络内幕》一书，学习了一下书中讲到的内核通知链方面的知识，写了一个读书笔记和一点代码来加深理解，希望能够对大家有一点帮助。内核通知链在网络方面得到了广泛的使用。

1.通知链表简介

　　大多数内核子系统都是相互独立的，因此某个子系统可能对其它子系统产生的事件感兴趣。为了满足这个需求，也

即是让某个子系统在发生某个事件时通知其它的子系统，Linux内核提供了通知链的机制。通知链表只能够在内核的子系统之间使用，而不能够在内核与用户空间之间进行事件的通知。

通知链表是一个函数链表，链表上的每一个节点都注册了一个函数。当某个事情发生时，链表上所有节点对应的函数就会被执行。所以对于通知链表来说有一个通知方与一个接收方。在通知这个事件时所运行的函数由被通知方决定，实际上也即是被通知方注册了某个函数，在发生某个事件时这些函数就得到执行。其实和系统调用signal的思想差不多。

2.通知链表数据结构

通知链表的节点类型为notifier_block，其定义如下：

```
struct notifier_block
{
    int (*notifier_call)(struct notifier_block *self, unsigned long, void *);
    struct notifier_block *next;
    int priority;
};
```

其中最重要的就是notifier_call这个函数指针，表示了这个节点所对应的要运行的那个函数。next指向下一个节点，即当前事件发生时还要继续执行的那些节点。

3.注册通知链

在通知链注册时，需要有一个链表头，它指向这个通知链表的第一个元素。这样，之后的事件对该链表通知时就会根据这个链表头而找到这个链表中所有的元素。

注册的函数是：

```
int notifier_chain_register(struct notifier_block **nl, struct notifier_block *n)
```

也即是将新的节点n加入到nl所指向的链表中去。

卸载的函数是：

```
int notifier_chain_unregister(strut notifier_block **nl, struct notifier_block *n)
```

也即是将节点n从nl所指向的链表中删除。

4.通知链表

当有事件发生时，就使用notifier_call_chain向某个通知链表发送消息。

```
int notifier_call_chain(struct notifier_block **nl, unsigned long val, void *v)
```

这个函数是按顺序运行nl指向的链表上的所有节点上注册的函数。简单地说，如下所示：

```
    struct notifier_block *nb = *n;
    while (nb)
    {
        ret = nb->notifier_call(nb, val, v);
        if (ret & NOTIFY_STOP_MASK)
        {
            return ret;
        }
        nb = nb->next;
    }
```

5.示例

在这里，写了一个简单的通知链表的代码。

实际上，整个通知链的编写也就两个过程：

首先是定义自己的通知链的头节点，并将要执行的函数注册到自己的通知链中。

其次则是由另外的子系统来通知这个链，让其上面注册的函数运行。

我这里将第一个过程分成了两步来写，第一步是定义了头节点和一些自定义的注册函数（针对该头节点的），第二步则是使用自定义的注册函数注册了一些通知链节点。分别在代码buildchain.c与regchain.c中。

发送通知信息的代码为notify.c。

代码1 buildchain.c

它的作用是自定义一个通知链表test_chain，然后再自定义两个函数分别向这个通知链中加入或删除节点，最后再定义一个函数通知这个test_chain链。

```
#include <asm/uaccess.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/notifier.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");

/*
 * 定义自己的通知链头结点以及注册和卸载通知链的外包函数
 */

/*
 * RAW_NOTIFIER_HEAD是定义一个通知链的头部结点，
 * 通过这个头部结点可以找到这个链中的其它所有的notifier_block
 */

static RAW_NOTIFIER_HEAD(test_chain);

/*
 * 自定义的注册函数，将notifier_block节点加到刚刚定义的test_chain这个链表中来
 * raw_notifier_chain_register会调用notifier_chain_register
 */

int register_test_notifier(struct notifier_block *nb)
{
 return raw_notifier_chain_register(&test_chain, nb);
}

EXPORT_SYMBOL(register_test_notifier);
int unregister_test_notifier(struct notifier_block *nb)
{
 return raw_notifier_chain_unregister(&test_chain, nb);
}
```

```
EXPORT_SYMBOL(unregister_test_notifier);
/*
 * 自定义的通知链表的函数，即通知test_chain指向的链表中的所有节点执行相应的函数
 */

int test_notifier_call_chain(unsigned long val, void *v)
{
 return raw_notifier_call_chain(&test_chain, val, v);
}

EXPORT_SYMBOL(test_notifier_call_chain);
/*
 * init and exit
 */

static int __init init_notifier(void)
{
 printk("init_notifier\n");
 return 0;
}

static void __exit exit_notifier(void)
{
 printk("exit_notifier\n");
}

module_init(init_notifier);
module_exit(exit_notifier);
```

代码2 regchain.c

该代码的作用是将test_notifier1 test_notifier2 test_notifier3这三个节点加到之前定义的test_chain这个通知链表上，同时每个节点都注册了一个函数。

```
#include <asm/uaccess.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/notifier.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");

/*
 * 注册通知链
 */
```

```
extern int register_test_notifier(struct notifier_block*);
extern int unregister_test_notifier(struct notifier_block*);

static int test_event1(struct notifier_block *this, unsigned long event, void *ptr)
{
 printk("In Event 1: Event Number is %d\n", event);
 return 0;
}

static int test_event2(struct notifier_block *this, unsigned long event, void *ptr)
{
 printk("In Event 2: Event Number is %d\n", event);
 return 0;
}

static int test_event3(struct notifier_block *this, unsigned long event, void *ptr)
{
 printk("In Event 3: Event Number is %d\n", event);
 return 0;
}

/*
 * 事件1，该节点执行的函数为test_event1
 */
 static struct notifier_block test_notifier1 =
{
 .notifier_call = test_event1,
};

/*
 * 事件2，该节点执行的函数为test_event1
 */


static struct notifier_block test_notifier2 =
{
 .notifier_call = test_event2,
};

/*
 * 事件3，该节点执行的函数为test_event1
 */

static struct notifier_block test_notifier3 =
{
 .notifier_call = test_event3,
};
```

```
/*
 * 对这些事件进行注册
 */

static int __init reg_notifier(void)
{

 int err;
 printk("Begin to register:\n");
 err = register_test_notifier(&test_notifier1);

 if (err)
 {
  printk("register test_notifier1 error\n");
  return -1;
 }

 printk("register test_notifier1 completed\n");
 err = register_test_notifier(&test_notifier2);
 if (err)
 {
  printk("register test_notifier2 error\n");
  return -1;
 }
 printk("register test_notifier2 completed\n");
 err = register_test_notifier(&test_notifier3);

 if (err)
 {
  printk("register test_notifier3 error\n");
  return -1;
 }
 printk("register test_notifier3 completed\n");
 return err;
}

/*
 * 卸载刚刚注册了的通知链
 */

static void __exit unreg_notifier(void)
{
 printk("Begin to unregister\n");
 unregister_test_notifier(&test_notifier1);
 unregister_test_notifier(&test_notifier2);
 unregister_test_notifier(&test_notifier3);
 printk("Unregister finished\n");
}
module_init(reg_notifier);
```

```
module_exit(unreg_notifier);
```

代码3 notify.c

该代码的作用就是向test_chain通知链中发送消息，让链中的函数运行。

```
#include <asm/uaccess.h>
#include <linux/types.h>
#include <linux/kernel.h>
#include <linux/sched.h>
#include <linux/notifier.h>
#include <linux/init.h>
#include <linux/types.h>
#include <linux/module.h>

MODULE_LICENSE("GPL");

extern int test_notifier_call_chain(unsigned long val, void *v);
/*
 * 向通知链发送消息以触发注册了的函数
 */

static int __init call_notifier(void)
{
 int err;
 printk("Begin to notify:\n");
/*
 * 调用自定义的函数，向test_chain链发送消息
 */
 printk("==============================\n");
 err = test_notifier_call_chain(1, NULL);
 printk("==============================\n");
 if (err)
  printk("notifier_call_chain error\n");
 return err;

}



static void __exit uncall_notifier(void)
{
 printk("End notify\n");
}

module_init(call_notifier);
module_exit(uncall_notifier);
```

Makefile文件

```
obj-m:=buildchain.o regchain.o notify.o
KERNELDIR:=/lib/modules/$(shell uname -r)/build
```

default:
 make -C $(KERNELDIR) M=$(shell pwd) modules

运行：
make
insmod buildchain.ko
insmod regchain.ko
insmod notify.ko
这样就可以看到通知链运行的效果了
下面是我在自己的机器上面运行得到的结果:
引用：
init_notifier
Begin to register:
register test_notifier1 completed
register test_notifier2 completed
register test_notifier3 completed
Begin to notify:
==============================
In Event 1: Event Number is 1
In Event 2: Event Number is 1
In Event 3: Event Number is 1